Parallel Processing Letters

THE RELATION BETWEEN DIAMOND TILING AND HEXAGONAL TILING

Tobias Grosser
INRIA and École Normale Supérieure, Paris
tobias.grosser@inria.fr

Sven Verdoolaege
INRIA, École Normale Supérieure and KU Leuven
sven.verdoolaege@inria.fr

Albert Cohen
INRIA and École Normale Supérieure, Paris
albert.cohen@inria.fr

P. Sadayappan
Ohio State University
saday@cse.ohio-state.edu

Received April 2014
Revised July 2014

Communicated by Guest Editors

ABSTRACT

Iterative stencil computations are important in scientific computing and more also in the embedded and mobile domain. Recent publications have shown that tiling schemes that ensure concurrent start provide efficient ways to execute these kernels. Diamond tiling and hybrid-hexagonal tiling are two tiling schemes that enable concurrent start. Both have different advantages: diamond tiling has been integrated in a general purpose optimization framework and uses a cost function to choose among tiling hyperplanes, whereas the greater flexibility with tile sizes for hybrid-hexagonal tiling has been exploited for effective generation of GPU code.

In this paper we undertake a comparative study of these two tiling approaches and propose a hybrid approach that combines them. We analyze the effects of tile size and wavefront choices on tile-level parallelism, and formulate constraints for optimal diamond tile shapes. We then extend, for the case of two dimensions, the diamond tiling formulation into a hexagonal tiling one, which offers both the flexibility of hexagonal tiling and the generality of the original diamond tiling implementation. We also show how to compute tile sizes that maximize the compute-to-communication ratio, and apply this result to compare the best achievable ratio and the associated synchronization overhead for diamond and hexagonal tiling.

Keywords: tiling, data-locality, polyhedra, parallelism, stencil

1. Introduction

Stencil computations are an important computational pattern in both scientific and engineering applications and they are becoming increasingly important in the embedded and mobile domain. Computational electromagnetics [23] and the numerical solution of partial differential equations [21] are common use cases of stencils in high performance computing, whereas image and video processing are driving forces in the embedded market. Even though manual and automatic optimizations of stencil computations have been addressed by numerous studies, the generation of efficient code remains a challenge especially for higher-dimensional stencils and for highly parallel platforms with multi-level hardware parallelism. With the increased use of parallel hardware in mobile markets as well as the foreseeable increase of 3D processing in upcoming embedded devices, a need emerges for solutions that facilitate the automatic generation of high-performance stencil codes for different devices.

For stencil computations, the tiling strategies that enable reuse along the time dimension have shown to be most efficient. The standard approach uses parallel wavefronts in a skewed index space. Skewed wavefronts reduce tile-level parallelism [15] and induce load-imbalanced prologue and epilogue phases. Split tiling [7, 15] and overlapped tiling [12, 15] address this problem by enabling concurrent start along one of the original iteration space dimensions. In other words, the tile schedule allows a wavefront of tiles parallel to one of the original dimensions of the index space to be executed in parallel. However, these two tiling techniques require either periodically alternating tile shapes or induce redundant computations. In contrast, the recently published diamond tiling [2] and hybrid-hexagonal tiling [8] schemes successfully obtain concurrent start without the need for redundant computations or multiple tile shapes.

Diamond tiling is a tiling strategy that uses a single n-dimensional parallelotope[a] that is constructed in such a way that it is possible to create a tiling where the number of tiles executable in parallel remains consistent throughout the computation, meaning that the tile schedule enables concurrent start. The advantages of diamond tiling are its integration in a general purpose compilation framework and the use of an adaptable cost function to determine tile shapes [2]. Hybrid hexagonal-classical tiling [8] is a tiling scheme that uses hexagonal tile shapes to enable concurrent start and to provide flexible tile size choices on one dimension. On the remaining dimensions it uses classical parallelogram tiling. The more domain specific formulation of hybrid-hexagonal tiling does not optimize tile shapes for a certain cost function, but always uses the most narrow dependence cone to derive the tile shape. On the other side, hybrid-hexagonal tiling has the advantage that it allows the adjustment of the time-tile height and the tile width along the spatial dimension independently. It also permits the creation of tiles with a flat top and can ensure that tiles not only have the same rational shape, but also identical integer

[a] A general term for what is known in 2D as parallelogram and in 3D as parallelepiped.

point placements by construction. Besides these advantages, there are also some open problems. Even though the diamond tiling paper generally explains how to derive tiling hyperplanes that enable concurrent start, a tile schedule that includes both the tile sizes as well as the parallel wavefront coefficients necessary to obtain concurrent start was not presented. This paper combines the two tiling strategies to benefit from the advantages of diamond and hexagonal tiling.

Our contributions are: a) an in-depth analysis of the constraints that diamond-tiling imposes on tile-sizes and wavefront coefficients, b) a formulation of conditions that ensure identical placement of integer points within the tiles, c) an extension of the original diamond tiling algorithm to a hexagonal tiling algorithm for 2 dimensional problems (1 time dimension, 1 space dimension), d) analysis of tile sizes to optimize the compute-to-communication ratio and synchronization overhead.[b] We note that while practically effective schemes based on diamond tiling [2] and hexagonal tiling [8] have been implemented, this paper focuses on a more abstract treatment. The development and evaluation of effective practical implementations based on the newly proposed ideas is beyond the scope of this short paper.

The paper is structured as follows. Section 2 revisits diamond tiling, providing insights on tile size and wavefront coefficient constraints, and discussing constraints and important properties of diamond tiles. We then introduce the unified hexagonal tiling scheme in Section 3 which includes a full formulation for two-dimensional tiling. Section 4 studies tile sizes that maximize the compute-to-communication ratio and compares the synchronizations induced by diamond and hexagonal tile shapes. We discuss related work in Section 5 and conclude in Section 6.

2. Diamond Tiling

The main contribution of diamond tiling [2] is the combination of affine transformations and a form of rectangular tiling that enables concurrent start. It is particularly effective on stencil computations. The idea of concurrent start is to ensure that the wavefront of tiles that are executed in parallel is aligned to a concurrent start hyperplane (normally an iteration space boundary) such that the number of tiles that are executed in parallel remains constant throughout the entire computation. This ensures that already at the beginning of the computation a sufficient amount of parallelism is available. Even though the name "diamond" suggests that the tile shapes are rhombi or rhombohedra (a.k.a. diamonds) and Figure 12 in Bandishti et al. [2] also uses edges of identical length, the tile shapes formed by diamond tiling are not restricted to diamonds, but can be more general parallelograms (parallelotopes in higher dimensions) as can be seen in Figure 3. However, some restrictions to the tile shape and sizes must be enforced to ensure that concurrent start is possible.

[b] This is an extended version of a paper presented at the HiStencils 2014 workshop [9]. The tile size analysis is entirely new content not appearing in [9].

2.1. *The Pluto Optimizer*

Diamond tiling was presented and implemented as an extension to Pluto [3], a general-purpose optimizer for data locality and parallelism. In contrast to other approaches that directly tile the iteration space (e.g., [7, 8]), the original Pluto tiling as well as diamond tiling are implemented as a two phase process. As a first step a program transformation is calculated that exposes sequences of loops (bands) that are tileable with rectangular tiles. In the second step a rectangular tiling is performed on these bands. Combined, this yields tiles with a possibly not rectangular, but parallelotope tile shape. There are several benefits of separating these two concerns. First, when calculating the parallel bands Pluto can and does perform other optimizations, e.g., data locality optimizations such as loop fusion. Second, tiling of the transformed program makes the tile shapes independent of the tiling hyperplanes, which makes the tiling easier to describe and analyze.

Pluto calculates program transformations on a polyhedral representation. In this representation the set of executed program statements (the iteration space) is modeled with a multi-dimensional integer set where each element represents an individual statement iteration. The execution order of elements of the iteration space is described by the schedule, an integer map that assigns a possibly multi-dimensional relative execution time to each element of the iteration space. Program transformations are performed by modifying the schedule. For a single statement and a k-dimensional execution time such a schedule has the form $S = \{\mathbf{x} \to (\mathbf{h}_0 \cdot \mathbf{x}, \ldots, \mathbf{h}_k \cdot \mathbf{x})\}$, where $\mathbf{x}$ is an element of the iteration space, $\mathbf{h}_i$, for $i \in \{0, \ldots, k\}$, are tiling hyperplanes represented by their normal vectors and $\mathbf{h}_i \cdot \mathbf{x}$ denotes the sum of the per element products of $\mathbf{h}_i$ and $\mathbf{x}$. The result of Pluto's first step are exactly these tiling hyperplanes, selected such that the distance between two statements that depend on each other is not only lexicographically nonnegative (needed for validity of the schedule), but also nonnegative at each individual dimension. As input, the algorithm takes an overapproximation of the pairs of statement instances that depend on each other, described using affine constraints. For the exact algorithm on how to select such hyperplanes, we refer to [3]. For the present paper, it is sufficient to understand that the all-nonnegative dependence vectors make rectangular tiling valid.

We present the Pluto rectangular tiling as a schedule only transformation which we believe is easier to understand than the actual Pluto transformation which modifies the iteration space as well. Conceptually, there should be no difference. Given a schedule S and a set of tile sizes $s_i, i \in \{0, \ldots, k\}$ a rectangularly tiled schedule of S consists of two partial schedules. The first one, S_t, is placed at the outer level and enumerates the tiles itself. This is called the tile schedule. The second one, S_p, is placed at the inner level and enumerates the points within each tile and is called the point schedule. We define $\{S_t = (x_0, \ldots, x_k) \to (\lfloor(\mathbf{h}_0 \cdot \mathbf{x})/s_0\rfloor, \ldots, \lfloor(\mathbf{h}_k \cdot \mathbf{x})/s_k\rfloor)\}$ and $S_p = S$. This tiled schedule may already expose parallelism, but exploiting it may involve a skewed wavefront schedule at the outermost tile dimension.

Then, such a wavefront schedule carries itself all dependences and ensures that the inner loops can be executed in parallel. This yields $S'_t = \{(x_0, \ldots, x_k) \rightarrow (\lambda_0 \lfloor (\mathbf{h}_0 \cdot \mathbf{x})/s_0 \rfloor + \cdots + \lambda_k \lfloor (\mathbf{h}_k \cdot \mathbf{x})/s_k \rfloor, \lfloor (\mathbf{h}_1 \cdot \mathbf{x})/s_1 \rfloor, \ldots, \lfloor (\mathbf{h}_k \cdot \mathbf{x})/s_k \rfloor)\}$ with $\lambda_i \in \mathbb{Z}_{\geq 0}, i \in \{0, \ldots, k\}$. The λ_i coefficients control the construction of different wavefronts. We call $\lambda_0 = \cdots = \lambda_k = 1$ the default wavefront coefficients. The hyperplanes computed by the original Pluto algorithm allow the formation of such a wavefront schedule, but those hyperplanes may not allow to form a wavefront schedule in the direction of a given concurrent start face (represented by its normal vector $\mathbf{f}$).

2.2. *The Diamond Tiling Extensions*

Diamond tiling [2] extends the Pluto algorithm in a way that ensures that for the tiling hyperplanes computed there always exist wavefront coefficients that yield concurrent start. From the original publication [2] we know that "a transformation enables tilewise concurrent start along a face $\mathbf{f}$ if and only if the tile schedule is in the same direction as the face and carries all inter-tile dependences". It also shows that "concurrent start along a face $\mathbf{f}$ can be exposed by a set of hyperplanes if and only if $\mathbf{f}$ lies strictly inside the cone formed by the hyperplanes, i.e., if and only if $\mathbf{f}$ is a strict conic combination of all the hyperplanes". For a concurrent start hyperplane $\mathbf{f}$, it finds tiling hyperplanes $\mathbf{h}_i$ such that the following equality holds:

$$m\mathbf{f} = \lambda_1 \mathbf{h}_1 + \cdots + \lambda_k \mathbf{h}_k \qquad \text{with } \lambda_i, m \in \mathbb{Z}_{\geq 0}. \tag{1}$$

The main focus of the diamond tiling paper is to prove the conditions necessary to ensure that the calculated hyperplanes can be used to construct a concurrent start schedule as well as to give an algorithm that actually calculates such hyperplanes. We therefore refer to this publication for details. One question that was explored less is under which conditions, especially for which tile sizes and for which wavefront coefficients, the rectangularly tiled schedule achieves concurrent start. Specifically, the paper does not investigate for which values of λ_i, s_j the following holds:

$$m\mathbf{x} \cdot \mathbf{f} = \lambda_0 \lfloor (\mathbf{h}_0 \cdot \mathbf{x})/s_0 \rfloor + \cdots + \lambda_k \lfloor (\mathbf{h}_k \cdot \mathbf{x})/s_k \rfloor \tag{2}$$

2.3. *Relation between Tile Sizes and Wavefronts*

Even though the diamond tiling yields tiling hyperplanes that allow concurrent start, to construct the full tile schedule the tile sizes s_i as well as the wavefront coefficients λ_i still need to be chosen. Choosing the correct values is important, not only to ensure that the tiles executed within the wavefront are started concurrently, but also to control the horizontal distance between neighboring tiles in the parallel wavefront and its ratio to the size of the tiles. We call this ratio the density of the schedule, a property important to understand the amount of computation that can be performed in parallel. Before suggesting good values, we explore the impact of different choices. Let us first consider a simple example with symmetric dependences:

```
for t:
  for i:
    A[t+1][i] = A[t][i-1] + A[t][i+1]
```

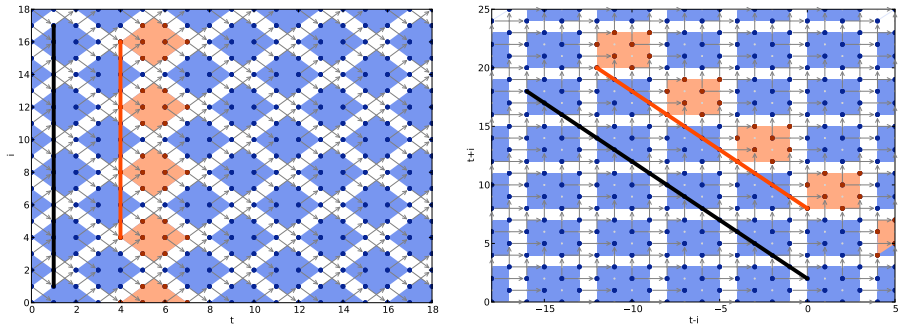


Fig. 1: Symmetric dependences & square tiling (original/transformed)

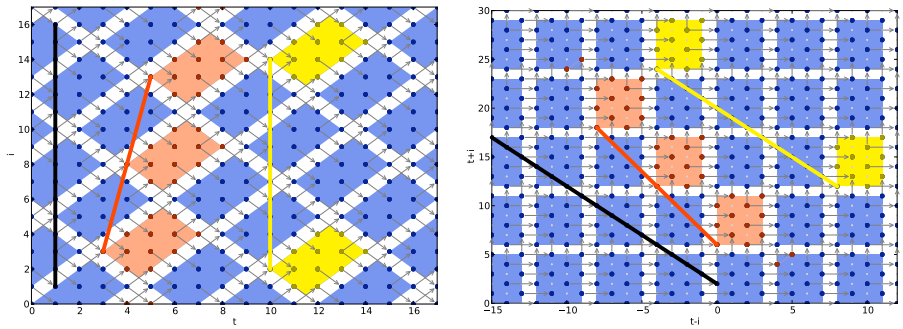


Fig. 2: Symmetric dependences & non-square tiling (original/transformed)

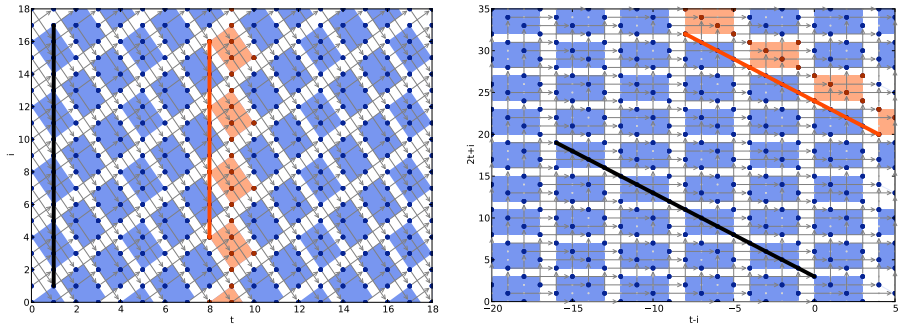


Fig. 3: Asymmetric dependences & square tiling (original/transformed)

Pluto's diamond tiling implementation[c] calculates for this kernel the transformation $\{(t,i) \rightarrow (t-i, t+i)\}$ and applies rectangular tiling in the transformed space. The default wavefront coefficients $\lambda_0 = \lambda_1 = 1$ are then used to enable

[c]Tested with version `0.10.0-50-g1a4ac17` from `git://repo.or.cz/pluto.git`

parallel execution. This results in the tile schedule $\{(t,i) \rightarrow (\lfloor (t-i)/s_0 \rfloor + \lfloor (t+i)/s_1 \rfloor, \lfloor (t+i)/s_1 \rfloor)\}$. The default square tile shapes ($s_0 = s_1$) yield both concurrent start as well as a high density of tiles. Figure 1 illustrates this for $s_0 = s_1 = 4$ with the tile wavefront highlighted in red and the concurrent start hyperplane highlighted in black. The two hyperplanes being parallel tells us that the tile wavefront has concurrent start. When different tile sizes are chosen for the two dimensions, the default wavefront no longer yields concurrent start. In Figure 2 we illustrate for $s_0 = 4, s_1 = 6$ that the default wavefront (red) is no longer parallel to the concurrent start hyperplane (black). Concurrent start is still possible with the non-default wavefront coefficients $\lambda_0 = 2, \lambda_1 = 3$, which yield the schedule $\{(t,i) \rightarrow (2\lfloor (t-i)/6 \rfloor + 3\lfloor (t+i)/4 \rfloor, \lfloor (t+i)/4 \rfloor)\}$. Unfortunately, a non-default wavefront causes a large loss in tile-level parallelism throughout the computation. This effect is illustrated by the yellow wavefront in Figure 2, which is parallel to the concurrent start hyperplane (black). Next we analyze a kernel with asymmetric dependences:

```
for t:
  for i:
    A[t+1][i] = A[t][i-1] + A[t][i+2]
```

Pluto derives from this kernel the transformation $\{(t,i) \rightarrow (t-i, 2t+i)\}$. This transformation combined with square tiling and the default wavefront coefficients allows concurrent start as shown in Figure 3 for $s_0 = s_1 = 4$. The reason for this, possibly surprising, result is that for a 2 dimensional stencil (1 space, 1 time) with dependence distance 1 in the time direction, the coefficient of the space dimension in the normal will always be ± 1. This ensures that when adding the two hyperplanes together their coefficients for the space dimension cancel out and we get again the concurrent start hyperplane. The default wavefront coefficients combined with square tile sizes therefore yield a concurrent start wavefront. As already found earlier, non-square tile sizes will prevent concurrent start with these coefficients.

Another interesting observation is that even though the rational tile shapes in Figure 3 are identical throughout the original iteration space, the set of contained integer points is not. The reason for this difference is that even though we use integral tile sizes in the transformed space, the borders may become non-integral in the original space. Varying integer point placements between tiles can cause problems due to additional conditions in the generated code. As a next step we consider a case of dependence distances with different lengths on the time dimension.

```
for t:
  for i:
    A[t+1][i] = A[t][i-1] + A[t-2][i+1]
```

For this kernel, the Pluto implementation derives the transformation $\{(t,i) \rightarrow (t-i, t+i)\}$. The same transformation was already chosen for the example illustrated

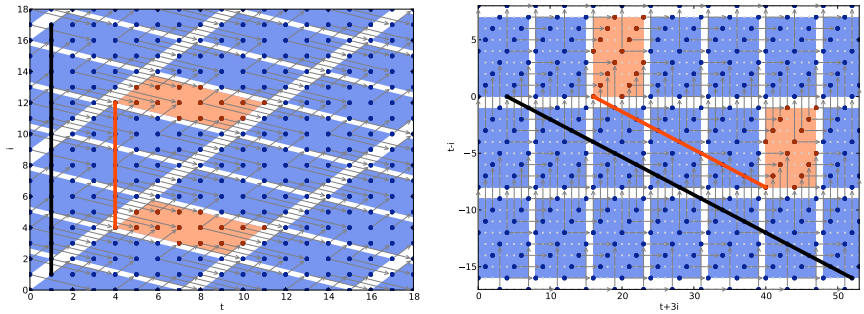


Fig. 4: Multiple time steps. Square tiles reduce parallelism. (original/transformed)

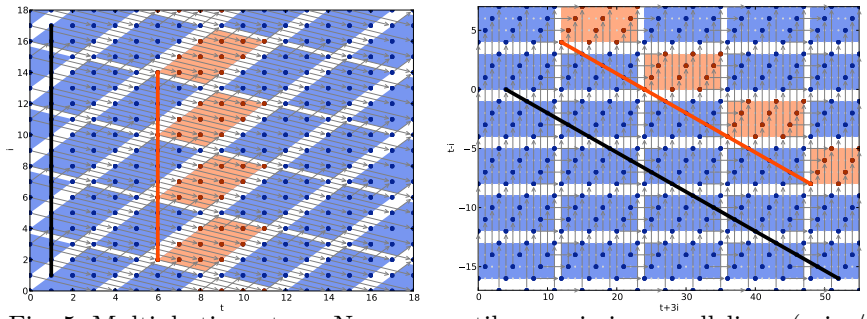


Fig. 5: Multiple time steps. Non-square tiles maximize parallelism. (orig./trans.)

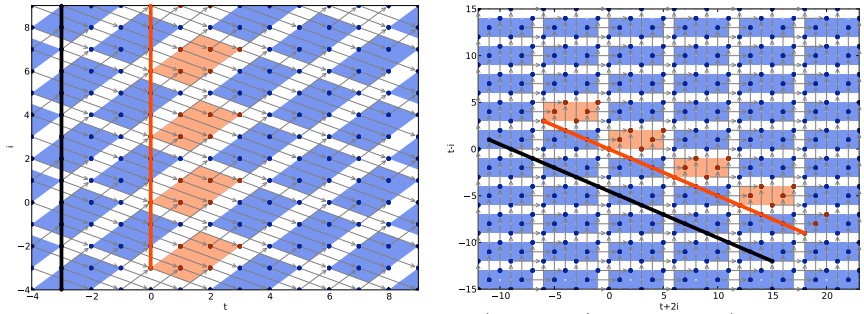


Fig. 6: Diamond tiling (original/transformed)

in Figure 1 and according to our understanding of the cost function in Pluto, this is in fact the transformation that the algorithm of [2] would choose. The resulting tiling yields 8 computations for a per-tile memory footprint of 3.

Another valid diamond tiling transformation is $\{(t, i) \to (t + 3i, t - i)\}$. The hyperplanes in this transformation are the ones hybrid-hexagonal tiling would read off directly from the dependence cone. Given a different cost function, Pluto may also choose this transformation. The interesting point here is, that the normal of the concurrent start hyperplane in the transformed space is not anymore (1,1), but rather (1,3). In this case, the standard square tiling illustrated in Figure 4 only yields concurrent start if, instead of the default wavefront coefficients, $\lambda_0 = 1, \lambda_1 = 3$ are chosen. As shown earlier, this severely reduces tile-level parallelism. On the other

hand, for the same memory footprint as before, this tiling executes 16 computations.

We can restore concurrent start with the default wavefront by using non-square tile sizes. Figure 5 shows a non-square tiling ($s_0 = 12, s_1 = 4$) which enables concurrent start, has maximal tile-level parallelism and reaches 12 computations for a memory footprint of three. We therefore prefer this tiling over the previous two.

2.4. *Optimal Tiles with Default Wavefront*

As seen in the previous section, the use of the default wavefront coefficients is necessary to ensure high tile-density. However, by itself this choice guarantees neither concurrent start nor a shared integer point placement for all tiles. As those properties are important, we present the conditions under which they can be reached.

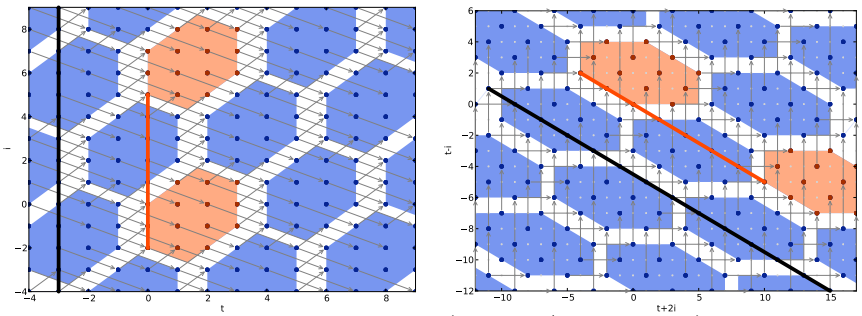


Fig. 7: Hexagonal-tiling (original/transformed)

We first explore the integer point placement. Forming the rows of matrix H from the tiling hyperplane normals $\mathbf{h}_i$, then tile sizes that are multiples of the determinant of H will ensure that all tiles have the same configuration of integer points since $\det(H) \cdot H^{-1}$ is an integer matrix. For example, the hyperplanes used in Figure 3 yield

$$H = \begin{pmatrix} 1 & -1 \\ 2 & 1 \end{pmatrix}$$

with $\det(H) = 3$. As $s_0 = s_1 = 4$ are not multiples of 3, the tiles may differ in integer point placement, as illustrated in the figure. On the other hand, tile sizes $s_0 = s_1 = 3$ would ensure a uniform integer placement across all tiles. The above condition is sufficient independently of the chosen wavefront schedule.

Next, we investigate the conditions on tile sizes to ensure concurrent start with the default default wavefront coefficients. Let $h_{x,0}$ be the first component of $\mathbf{h}_x$ and $h_{x,1}$ the second. The default wavefront then is $\lfloor (h_{0,0}t + h_{0,1}i)/s_0 \rfloor + \lfloor (h_{1,0}t + h_{1,1}i)/s_1 \rfloor$. Now, to achieve concurrent start, we need to ensure that the default wavefront schedule only depends on the time dimension t and that all space dimensions (i.e., i) are eliminated. This is true under the condition $s_0/|h_{0,1}| = s_1/|h_{1,1}|$.

Note that the wavefront may still depend on the fractional part of the space dimension, but this only results in a variation within a fixed range, independently of the size of the domain. We can see that in Figure 1, where we reach concurrent start for the default wavefront, this condition holds with $4/1 = 4/1$. On the other hand, when changing the tile sizes to $s_0 = 4$ and $s_1 = 6$ as in Figure 2, the previous condition turns into $4/1 = 6/1$ and concurrent start is not possible with the default wavefront. The above shows that to obtain concurrent start the two tile sizes cannot be chosen independently, but need to be scaled together. To make this more clear we introduce a new variable s which can be chosen freely and which is then used to define $s_0 = s|h_{0,1}|$ and $s_1 = s|h_{1,1}|$ such that concurrent start is obtained. Interestingly, this condition of a single parameter defining the tile size is exactly what is required by the parametric tiling approach of Iooss et al. [14].

3. Unified Diamond and Hexagonal Tiling

In this section we present for a two-dimensional iteration space an extended formulation of diamond tiling which allows the creation of hexagonal tiles. The hexagonal tiles calculated are similar to those presented in [8], but are not identical in shape.

To obtain such a schedule we start from the diamond tiling approach, which means we first calculate a set of tiling hyperplanes, transform the index space with these hyperplanes and then apply rectangular tiling in the transformed space. We then (optionally) transform the rectangular tiling by "stretching" the rectangular tiles along the concurrent start hyperplane. The stretched rectangular tiles in the transformed space form hexagonal tiles in the original space. As a result we have a single schedule that describes diamond tiling, if tiles are stretched by a vector of length zero, and hexagonal tiling, if they are stretched by a non-zero-length vector.

In the following description, we assume that the tiling hyperplanes h_0, h_1 are computed by the diamond tiling algorithm as described in [2]. We focus on the description of the (possibly) stretched tiling scheme in the transformed space. As input for the stretched tiling scheme, we take the tile sizes s_0, s_1 as well as a vector $\mathbf{v} = (v_0, v_1)$, which is parallel to the concurrent start hyperplane (in the transformed space). We also require this hyperplane to have a normal $\mathbf{n} = (n_0, n_1)$ that is strictly positive in all components, as guaranteed by the algorithm of [2].

We first model diamond tiling using a standard 2D rectangular tiling in the transformed space. In this tiling the symbols s_0, s_1 define the tile sizes along the dimensions d_0, d_1 while T_0, T_1 are the resulting tile schedule dimensions (we ignore the point schedule dimensions, as this mapping is not interesting for this discussion). The following map describes such a rectangular tiling.

$$\{(d_0, d_1) \to (T_0, T_1) \mid s_0 T_0 \leq d_0 < s_0(T_0 + 1) \wedge s_1 T_1 \leq d_1 < s_1(T_1 + 1)\} \quad (3)$$

Our goal is to achieve and maintain concurrent start using the default wavefront. Consequently s_0 and s_1 cannot be chosen freely (see Section 2.4). We require the user to choose tile sizes that ensure concurrent start. Figure 6 illustrates the above

rectangular tiling using the transformation $\{(t,i) \to (t+2i, t-i)\}$, as well as the tile sizes $s_0 = 6, s_1 = 3$. The red tiles show the concurrent start wavefront.

Starting from this rectangular tiling we want to stretch the contained tiles by a vector $\mathbf{v}$ with components v_0, v_1, where $\mathbf{v}$ is parallel to the concurrent start hyperplane. More formally, we want to compute for each tile represented by T the set of points contained in a new tile T'. T' is defined as the Minkowsky sum $T + V$, where $V = \{t\mathbf{v} \mid 0 \leq t \leq 1\}$ and the Minkowsky sum of is defined as $A + B = \{\mathbf{a} + \mathbf{b} \mid \mathbf{a} \in A,\ \mathbf{b} \in B\}$. In addition we translate the new tiles to again reach a space filling tiling. In principle, $\mathbf{v}$ can have either of two possible directions, but to simplify the schedule formulation we choose $\mathbf{v}$ such that $v_0 < 0 \wedge v_1 > 0$. Figure 7 shows a stretching as we obtain it for $\mathbf{v} = (-4, 2)$ and $\mathbf{n} = (1, 2)$.

Before we implement the actual stretching, we first add two additional constraints to each tile. The first one bounds each tile at its lexicographic minimal point with the concurrent start hyperplane, the second one bounds each tile at its lexicographic maximal point with the same (but translated) hyperplane. We implement the lower boundary by placing the hyperplane at the origin and by offsetting it for each tile according to the tile sizes. To offset the tile along d_0 we adjust the right hand side of the lower bound by $n_0 s_0 T_0$ and $n_1 s_1 T_1$. The upper boundary is implemented by reversing the lower hyperplane. The location of the upper hyperplanes for tile (T_0, T_1) is the origin of tile $(T_0 + 1, T_1 + 1)$.

$$\begin{aligned}\{(d_0, d_1) \to (T_0, T_1) \mid & \\ & s_0 T_0 \leq d_0 < s_0(T_0 + 1) \\ \wedge\ & s_1 T_1 \leq d_1 < s_1(T_1 + 1) \\ \wedge\ & n_0 s_0 T_0 + n_1 s_1 T_1 \leq n_0 d_0 + n_1 d_1 \\ \wedge\ & n_0 d_0 + n_1 d_1 < n_0 s_0(T_0 + 1) + n_1 s_1(T_1 + 1)\}\end{aligned} \tag{4}$$

As a last step, we now stretch the tiles along $\mathbf{v}$. This requires us to increase the size of the rectangular tiles by v_0 in the d_0 dimension and v_1 in the d_1 dimension. We also account for the shifted positions of the rectangular tiles by adding some offsets o_0, o_1 to the upper and lower tile boundaries that will be derived later in this section. Finally, we adjust the locations of the concurrent start planes by using $c_0 = n_1(s_0 + v_0) + n_0 v_1$ and $c_1 = n_1(s_1 + v_1) + n_0 v_0$.

$$\begin{aligned}\{(d_0, d_1) \to (T_0, T_1) \mid \exists o_0 = -v_0 T_0 + v_0 T_1, o_1 = -v_1 T_0 + v_1 T_1 : & \\ s_0 T_0 + o_0 + v_0 \leq d_0 < s_0(T_0 + 1) + o_0 & \\ \wedge\ s_1 T_1 + o_1 \leq d_1 < s_1(T_1 + 1) + v_1 + o_1 & \\ \wedge\ c_0 T_0 + c_1 T_1 \leq n_0 d_0 + n_1 d_1 & \\ \wedge\ n_0 d_0 + n_1 d_1 < c_0(T_0 + 1) + c_1(T_1 + 1)\} &\end{aligned} \tag{5}$$

Figure 8 illustrates the last step in detail. On the left side, the red tiles are the original square tiles (0,0), (1,0) and (1,1), each of size 6×4. On the right side, the same tiles have been stretched along $\mathbf{v}$. The rectangular tile shapes have been

extended by 4 along d_0 and by 2 along d_1 resulting in the light blue tile shapes (the dark blue tile shapes illustrate the contained integer points). We can also see that the position of the red tile shape of tile (0,0) has not moved. However, when going one step up to tile (1,0) which means increasing the tile number T_0 by one, we offset the tile by $-v_0$ along d_0 as well as $-v_1$ along d_1. Similarly, when going from tile (1,0) to tile (1,1) which means increasing the tile number T_1 by one, we offset the tile by v_0 along d_0 and v_1 along d_1. Combined this yields the offset $o_0 = -v_0T_0 + v_0T_1$ for d_0 and $o_1 = -v_1T_0 + v_1T_1$ for d_1. The new values c_0 and c_1 do now also take into account the offset of the plane. When varying T_0 we now do not only need to take the vertical tile size s_0 into account, but in addition we include the additional vertical offset v_0 as well as the changed horizontal offset v_1. To support concurrent start hyperplanes of different orientations such offsets are scaled by the relevant components of $\mathbf{n}$. The corresponding changes have been added when adjusting c_1.

A very important observation is that tiles (T_0, T_1) as well as (T_0+1, T_1+1) have overlapping rectangular parts. However, the concurrent start hyperplanes added at the position of $\mathbf{v}$ ensure that tiles are non-overlapping and still tile the full space. Also, as stretching and translation are carried along the concurrent start hyperplane, no dependences are violated. Finally, if the previous tiling had concurrent start, stretching along the concurrent start hyperplane preserves this property.

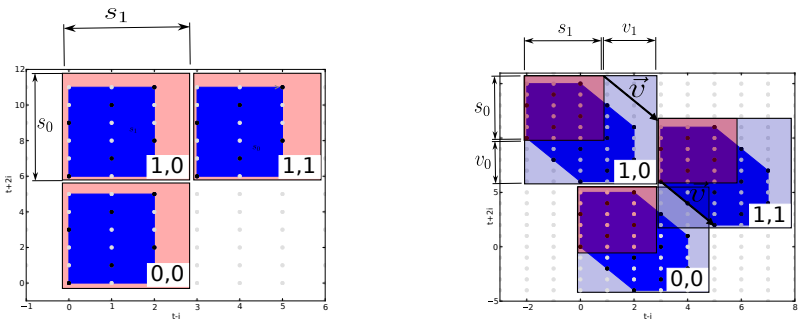


Fig. 8: The stretching in the transformed space (unstretched/stretched)

4. Tile Sizes that Maximize Compute/Communication

In this section we analyze how to maximize the compute-to-communication ratio and show how the tiling strategy choice affects both compute-to-communication ratio and synchronization overhead. Using a simple theoretical compute model, we derive for diamond and hexagonal tiles basic characteristics such as the number of operations executed, the amount of communication, the usage of local memory as well as the amount of synchronization needed for parallelism. We use these characteristics to understand the effectiveness of the different tile shapes. We use the following 3-point heat stencil as illustrative compute pattern.

```
for t:
    for i:
S:        A[(t+1)%2][i] =  A[t%2][i-1] + A[t%2][i] + A[t%2][i+1];
```

For this analysis the iteration space boundaries are not of relevance, but we obtain the dependences $D = \{ \mathtt{S}(t,i) \rightarrow \mathtt{S}(t+1,i') \mid i'-1 \leq i \leq i'+1\}$ (transitively covered dependences removed), which are for this specific example code identical to the set of flow dependences ($D_{\mathrm{F}} = D$). Furthermore, we obtain mappings $A_{\mathrm{R}} = \{S(t,i) \rightarrow A(t \bmod 2, i') \mid i-1 \leq i' \leq i+1\}$ and $A_{\mathrm{W}} = \{S(t,i) \rightarrow A((t+1) \bmod 2, i)\}$ specifying for each statement instance the data locations read from and written to. From this information we compute, e.g., using Pluto, the concurrent start tiling hyperplanes $t+i$ and $t-i$. However, instead of using now the full unified schedule (5) to tile the space, we derive a description of the set of iterations that belong to the single tile that will be placed at the origin. We use this tile as our tile shape model. To obtain its description we take the unified schedule from (5) and extract the set of input values that correspond to $T_0 = T_1 = 0$:

$$\begin{aligned}\{(d_0,d_1) \mid\ & v_0 \leq d_0 < s_0 \\ \wedge\ & o_1 \leq d_1 < s_1 + v_1 \\ \wedge\ & 0 \leq n_0 d_0 + n_1 d_1 \\ \wedge\ & n_0 d_0 + n_1 d_1 < n_1(s_0+v_0) + n_0 v_1 + n_1(s_1+v_1) + n_0 v_0\}\end{aligned} \tag{6}$$

We then set $(n_0, n_1) = (1,1)$ according to the tiling hyperplanes we computed for our example and we set $(s_0, s_1) = (T, T)$ and $(v_0, v_1) = (-B, B)$ to introduce two variables T and B that control the size of the tile.

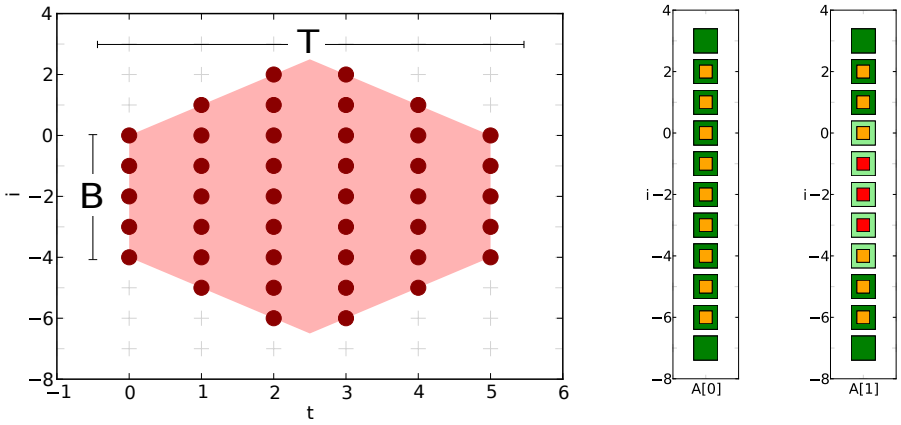
$$\{(d_0,d_1) \mid -B \leq d_0 < T \wedge 0 \leq d_1 < T + B \wedge 0 \leq d_0 + d_1 \wedge d_0 + d_1 < 2T\} \tag{7}$$

As a final step, we use our tiling hyperplanes to translate this tile shape description back into the original space.

$$\begin{aligned}I = \{S(t,i) \mid\ & -B \leq (t+i) < T\ \wedge\ 0 \leq (t-i) < T + B \\ & \wedge\ 0 \leq (t+i) + (t-i)\ \wedge\ (t+i) + (t-i) < 2T\}\end{aligned} \tag{8}$$

The result is I, a parametric tile shape description. It can be expressed with affine constraints despite the fact that a parametric description of all tiles can not. The tile shape we obtained has the form of a hexagon. In the illustration in Figure 9a we can see the two parameter that define its size: T defines the width of the tile along the time dimension t, and B defines the distance by which the tile is extended along the space dimension i. When $B = 0$ the tile shape degenerates to a diamond shape.

We now present the data access model we use for the computations in this tile. Figure 10 illustrates the first steps. At the first time step ($t = 0$), five elements are computed. Seven elements are read from array A[0] (dark small squares) and the resulting five elements are written into the scratchpad (dark small triangles). At the second time step ($t = 1$) seven elements are computed. We already have seven

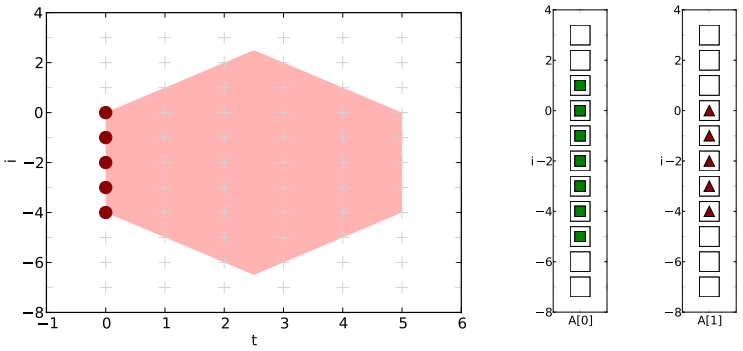


(a) Iteration space (b) Accessed data

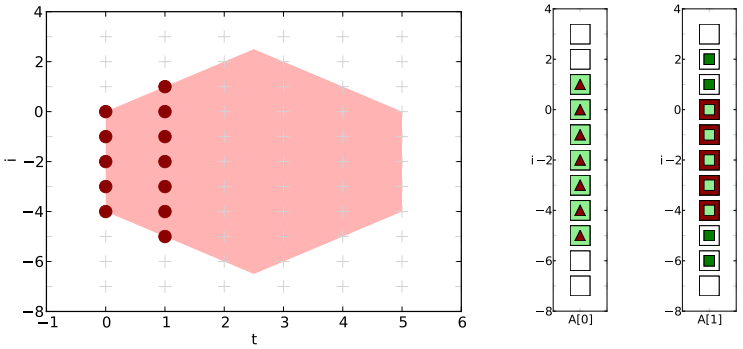
Fig. 9: 1D hexagonal tiling ($T = 6, B = 4$)

elements in our scratchpad that have been previously read (light large squares) and five elements that have been previously computed (dark large squares). For the computation we need to read nine elements from array A[1] and seven elements are stored back to the scratchpad. From the nine elements read five can be read from the scratchpad (light small squares). Similarly, when computing the nine new elements of the third time step ($t = 2$), from the eleven elements read, seven have been computed in a previous time step. Going further in time we see, that with exception of $t = 3$ all time steps in the decreasing phase ($t = 4$ and $t = 5$) only read data that has been computed in previous time steps. Combining the memory accesses we can derive the set of data elements read (large squares) and the set of data elements written (small squares) in a tile. For the data elements read, we can distinguish between the data elements that at least once need to be read from external memory (dark large square) and those that can directly be read from the local scratchpad (light large square). Similarly, for the data elements written we distinguish between the data elements that will be used by later tiles (light small square) and the ones that are only needed in this very tile, but do not need to be written out (dark small square). This is illustrated in Figure 9b.

We compute the following per tile characteristics: the number of compute operations (O), the number of data locations read (R_{ALL}), the number of data locations read not considering data previously computed in the same tile (R_{REDUCED}), the number of data locations written to (W_{ALL}), the number of data locations written to and needed by later computations (W_{REDUCED}), the number of synchronization steps (S), as well as the footprint (F), i.e., the set of all data locations accessed. As these characteristics correspond to the number of integer points in certain sets, we can derive parametric closed form expressions by using barvinok [27] to count



(a) Iteration Space (t=0) (b) Data



(c) Iteration Space (t=1) (d) Data

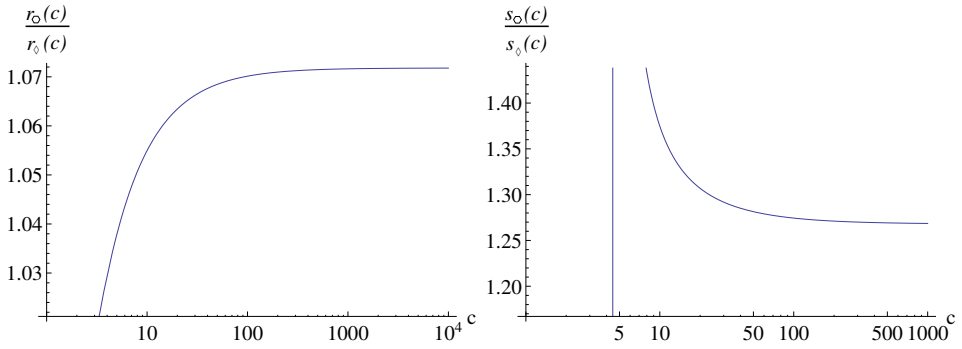
Fig. 10: The first two time steps of 1D hexagonal tiling ($T = 6, B = 4$)

these points. We perform the following computations: $O = |I|$, $R_{\text{ALL}} = |A_{\text{R}}(I)|$, $W_{\text{ALL}} = |A_{\text{W}}(I)|$, $F = |A_{\text{R}}(I) \cup A_{\text{W}}(I)|$. There are only two slightly more complicated computations. First, $R_{\text{REDUCED}} = |A_{\text{W}}(D_{\text{F}}^{-1}(I) \setminus I)|$, the set of data locations that are read from within the tile without any statement in the tile having written to them previously. We compute this as the set of data locations written from statements that are at the origin of a flow dependence that ends in the tile, but are themselves not within the tile. Second, $W_{\text{REDUCED}} = |A_{\text{W}}(D_{\text{F}}^{-1}(U \setminus I) \cap I)|$, the set of data locations written to inside the tile and later read by a statement instance outside of the tile without being overwritten in between with data computed after the execution of the tile finished. We compute this starting from the universal set from which we remove the statement instances inside the tile to obtain the instances that are not in this tile. By applying the reverse flow dependences and intersecting with the tile again, we obtain the set of instances in the tile that write values that are used outside of the tile. We get the corresponding data locations by applying A_{W}.

The results[d] we obtain from barvinok are the following formulas:

$$O(T,B) = \frac{1}{2}T^2 + TB\,, \qquad S(T) = T - 1$$
$$R_{\text{ALL}}(T,B) = 2T + 2B + 2\,, \qquad W_{\text{ALL}}(T,B) = 2T + 2B - 2$$
$$R_{\text{REDUCED}}(T,B) = 2T + B + 1\,, \qquad W_{\text{REDUCED}}(T,B) = 2T + B - 1$$
$$F(T,B) = 2T + 2B + 2$$

As a next step we compute certain properties. We start with the compute-per-read ratio $r(T,B) = O(T,B)/R_{\text{ALL}}(T,B)$ and maximize it for a fixed "cache size" c, i.e., the number of data elements that fit in the scratchpad. For this we formulate an optimization problem and use the computer algebra system Mathematica [13] to solve it symbolically. The result is $\max_{T,B} r(T,B)\big|_{F\le c} = \frac{(c-2)^2}{8c}$ and $\arg\max_{T,B} r(T,B)\big|_{F\le c} = (1/2(c-2), 0)$. We make two observations here: first, $r(c)$ increases linearly with the available cache size; second, the optimal tile shape has $B = 0$. This means it degenerates to a diamond.



(a) Compute-to-read (b) Compute-to-sync

Fig. 11: Relation between hexagonal and diamond tiling ratios

When maximizing the ratio $r'(T,B) = O(T,B)/R_{\text{REDUCED}}(T,B)$ we obtain: $r'_{\bigcirc}(c) = \max_{T,B:F(T,B)\le c} r'(T,B) = c - \frac{1}{2}\sqrt{c(3c-4)} - 1$ at $T_{\bigcirc} = \frac{1}{2}\left(\sqrt{21c^2 - 4\left(3\sqrt{c(3c-4)}+7\right)c + 8\sqrt{c(3c-4)} + 4} + 2c - \sqrt{c(3c-4)} - 2\right)$ and $B_{\bigcirc} = c - \frac{1}{2}\sqrt{c(3c-4)} - 1$. We see that the optimal ratio is obtained with $B \neq 0$, a hexagonal tile shape not degenerated to a diamond. We now derive the optimal ratio with the additional constraint $B = 0$, which limits our search to diamond shaped tiles. We obtain $r'_{\diamond}(c) = \max_{T:F(T,0)\le c} r'(T,0) = \frac{(c-2)^2}{8c}$ at $T_{\diamond} = 1/2(c-2)$. We can see that both $r'_{\bigcirc}$ and $r'_{\diamond}$ increase linearly with c, showing that hexagonal tiles are more efficient than diamond tiles. To understand how much more efficient

[d]The formulas have been simplified under the assumptions $T \bmod 2 = 0 \wedge B \bmod 2 = 0$.

they are, Figure 11a captures the evolution of $r'_{\circ}/r'_{\diamond}$, making clear that as soon as the scratchpad can hold more than about 100 data elements, hexagonal tiles yield a ratio O/R_{REDUCED} that is more than 7% higher than diamond tiles. As W_{ALL} and W_{REDUCED} only differ by a small constant from the corresponding read properties, for sufficiently large scratchpads ratios similar to the ones we computed for the read properties will be obtained and similar conclusions can be taken.

Another interesting property is the amount of synchronization steps necessary per tile. For this we compute $s_{\circ} = O(T_{\circ}, B_{\circ})/S(T_{\circ})$ and $s_{\diamond} = O(T_{\diamond}, B_{\diamond})/S(T_{\diamond})$ which give an idea of how much computation can be performed for a certain number of synchronization steps. Figure 11b illustrates $s_{\circ}/s_{\diamond}$ to compare the hexagonal and diamond tiling strategies. We see that the graph quickly converges to a value above 26%. This means 26% more computations can be executed for the same amount of synchronization when using hexagonal tiling. As these results are computed on a model, they obviously do not directly translate to a specific piece of hardware, but they show that for hardware where in-tile synchronization is costly hexagonal tiling can result in significant improvements.

5. Related Work

Aside from the already discussed diamond and hybrid-hexagonal tiling [2, 8], there has been a lot of successful research in generating code to efficiently perform stencil computations.

Christen et al. [4] propose the PATUS stencil compiler, a system which given the description of a parameterizable code generation and parallelisation strategy as well as the specification of a specific stencil generates efficient code using autotuning. Han et al. [10] develop with PADS pattern-based optimization of stencil codes for CPUs and GPUs using a proposed extension to OpenMP. Similar to PATUS, code generation strategies can be provided by the user. The different strategies provided together with PATUS and PADS are to our understanding all space tiling strategies which do not exploit reuse along the time dimension. Datta et al. [6, 5] develop an optimization and auto-tuning framework for stencil computations, targeting multi-core systems, NVIDIA GPUs, and Cell SPUs. Membarth at al. [17] discuss with HIPAcc a source to source compiler for image processing kernels that can use abstract hardware models to generate GPU code that addresses the different performance characteristics of various GPU models. Similarly to the previous systems, the last two approaches do not consider time tiling.

Tang et al. present with Pochoir [24], a domain-specific C++ framework that uses cache-oblivious parallelograms to generate C++ code. Pochoir takes advantage of thread level parallelism through the Cilk runtime system. Henretty et al. [11] use a DSL-based approach relying on different versions of split tiling in combined with a data layout transformation to generate efficient SIMD CPU code. Strzodka [22] uses an in-tile wavefront traversal technique to achieve efficient cache use even with tile sizes larger than the available cache memory. All these approaches generate ef-

ficient CPU code. Then, there are a set of general optimizers. PPCG [26] generates parallel CPU and GPU code using classical (time) tiling. It relies on affine transformations to extract parallelism and improve locality, using a variant of the Pluto algorithm [3]. Reservoir Labs' R-Stream is also a reference polyhedral compiler targeting GPUs [16, 25]. Par4All [1] is an open source parallelizing compiler developed by Silkan targeting multiple architectures. The compiler is not based on the polyhedral model, but uses abstract interpretation for array regions, performing powerful inter-procedural analysis on the input code. Finally, there are tools that generate efficient GPU code. Here Holewinski's Overtile [12] and Grosser's split tiling [7] compilers represent, besides [8], the state-of-the-art for the automatic generation of efficient GPU code relying on overlapped and split tiling, respectively.

Computing optimal tile sizes has a long history [20]. Our approach of symbolically computing a closed form solution using a computer algebra system is closely related to the positivity based tile size selection framework of Renganarayana and Rajopadhye [19], with the difference that we derive parts of our cost model by counting integer points. Also the use of such an approach to compare hexagonal and diamond tiling schemes is new. Orozco et al. [18] use the dependency graph to find the optimal tile shape for stencil computations. They conclude, just like us, that diamond tiling is the most efficient tile shape, if the full read set R_{ALL} is considered. They do not give information about optimal tile shapes when considering the ratio between computation and reduced read sets (R_{REDUCED}) or computation and synchronization overhead.

6. Conclusion

We presented a formulation of hexagonal tiling that combines the benefits of diamond tiling and hybrid-hexagonal tiling. For diamond tiling, we formalized conditions on tile sizes and wavefront coefficients to ensure concurrent start among tiles. We also formulated a condition to ensure the same integer point placement across all tiles. And most importantly, we extended the original diamond tiling algorithm to hexagonal tiles. Using a simple cost model, we performed a comparative analysis of the compute-to-communication ratio of diamond and hexagonal tiling schemes as a function of tile sizes and characterized the optimal aspect ratio for hexagonal tiles and the benefit over diamond tiles. The analyses and hybrid approaches developed in this paper could serve as the basis for practical implementations that improve on the current state-of-the art in tiling stencil computations.

Acknowledgments This work greatly benefited from regular discussions with Uday Bondhugula. It was partly funded by a Google European Fellowship in Efficient Computing, by the European FP7 project CARP id. 287767, the COPCAMS ARTEMIS project, and award 0926688 from the U.S. NSF.

References

[1] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, et al. Par4All: From convex array regions to heterogeneous computing. In *IMPACT*, 2012.

[2] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *ACM Supercomputing Conf.*, 2012.

[3] Uday Bondhugula, J. Ramanujam, and et al. PLuTo: A practical and fully automatic polyhedral program optimization system. In *PLDI*, 2008.

[4] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, pages 676–687. IEEE, 2011.

[5] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine A. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[6] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of SC '08*, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[7] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *GPGPU-6*, pages 24–31. ACM, 2013.

[8] Tobias Grosser, Albert Cohen, Sven Verdoolaege, P. Sadayappan, and Justin Holewinski. Hybrid hexagonal/classical tiling for GPUs. In *CGO*, 2014.

[9] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P. Sadayappan. The relation between diamond tiling and hexagonal tiling. HiStencils, Vienna, Austria, 2014.

[10] Dongni Han, Shixiong Xu, Li Chen, and Lei Huang. Pads: A pattern-driven stencil compiler-based tool for reuse of optimizations on gpgpus. In *ICPADS*, pages 308–315, 2011.

[11] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *International Conference on Supercomputing (ICS)*. ACM, 2013.

[12] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*. ACM, 2012.

[13] Wolfram Research Inc. Mathematica 9.0, 2012.

[14] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. Cart: Constant aspect ratio tiling. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, January 2014.

[15] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, 2007.

[16] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *PGPGPU*, New York, NY, USA, 2010.

[17] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Kostler. Towards domain-specific computing for stencil codes in hpc. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1133–1138. IEEE, 2012.

[18] Daniel Orozco, Elkin Garcia, and Guang Gao. Locality optimization of stencil applications using data dependency graphs. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2011.

[19] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. Positivity, posynomials and tile size selection. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[20] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical report, 1990.

[21] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 2004.

[22] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and H Seidel. Cache accurate time skewing in iterative stencil computations. In *Parallel Processing (ICPP)*, 2011.

[23] A. Taflove. *Computational electrodynamics: The Finite-difference time-domain method*. Artech House, 1995.

[24] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *SPAA*. ACM, 2011.

[25] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT*, Paris, France, January 2012.

[26] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM TACO*, 9(4):54, 2013.

[27] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.